Accelerating fluid–solid simulations (Lattice-Boltzmann & Immersed-Boundary) on heterogeneous architectures

Pedro Valero-Lara, Francisco D. Igual, Manuel Prieto-Matías, Alfredo Pinelli, Julien Favier

Accelerating fluid–solid simulations (Lattice-Boltzmann & Immersed-Boundary) on heterogeneous architectures

Pedro Valero-Lara [a,*], Francisco D. Igual [b], Manuel Prieto-Matías [b], Alfredo Pinelli [c], Julien Favier [d]

[a] BCAM – Basque Center for Applied Mathematics, Bilbao, Spain
[b] Departamento de Arquitectura de Computadores y Automaática, Universidad Complutense de Madrid, Madrid, Spain
[c] School of Engineering and Mathematical Sciences, City University London, London, United Kingdom
[d] Aix Marseille Université, CNRS, Centrale Marseille, M2P2 UMR 7340, 13451 Marseille, France

ARTICLE INFO

Article history:
Received 13 October 2014
Received in revised form 26 May 2015
Accepted 2 July 2015
Available online 14 July 2015

Keywords:
Parallel computing
Computational fluid dynamics
Fluid–solid interaction
Lattice-Boltzmann method
Immersed-Boundary method
Heterogeneous computing

ABSTRACT

We propose a numerical approach based on the Lattice-Boltzmann (LBM) and Immersed Boundary (IB) methods to tackle the problem of the interaction of solids with an incompressible fluid flow, and its implementation on heterogeneous platforms based on data-parallel accelerators such as NVIDIA GPUs and the Intel Xeon Phi. We explain in detail the parallelization of these methods and describe a number of optimizations, mainly focusing on improving memory management and reducing the cost of host-accelerator communication. As previous research has consistently shown, pure LBM simulations are able to achieve good performance results on heterogeneous systems thanks to the high parallel efficiency of this method. Unfortunately, when coupling LBM and IB methods, the overheads of IB degrade the overall performance. As an alternative, we have explored different hybrid implementations that effectively hide such overheads and allow us to exploit both the multi-core and the hardware accelerator in a cooperative way, with excellent performance results.

1. Introduction

The dynamics of a solid in a flow field is a research topic with a growing interest in many scientific communities. It is intrinsically interdisciplinary (structural mechanics, fluid mechanics, applied mathematics, ...) and covers a broad range of applications (e.g. aeronautics, civil engineering, biological flows, etc.). The number of works in this field is rapidly increasing, which reflects the growing importance of studying the dynamics in the solid–fluid interaction [1–4]. Most of these simulations are compute-intensive and benefit from high performance computing systems. However, they also exhibit an irregular and dynamic behaviour, which often leads to poor performance when using emerging heterogeneous systems equipped with many-core accelerators.

Many computational fluid dynamic (CFD) applications and software packages have already been ported and redesigned to exploit heterogeneous systems. These developments have often involved major algorithm changes since some classical solvers may turned out to be inefficient or difficult to tune [5,6]. Fortunately, other solvers are particularly well suited for GPU acceleration and are able to achieve significant performance improvements. The Lattice Boltzmann method (LBM) is one of those examples thanks to its inherently data-parallel nature. Certainly, the computing stages of LBM are amenable to fine grain parallelization in an almost straightforward way. This fundamental advantage of LBM has been consistently confirmed by many authors [7–9,11], for a large variety of problems and computing platforms.

In this paper, we explore the benefits of LBM solvers on heterogeneous systems. Our target application is an integrated framework that uses the Immersed Boundary (IB) method to simulate the influence of a solid immersed in a incompressible flow [11]. Some recent works that cover subjects closely related with our contribution are [9] and [12]. In [9], authors presented an efficient 2D implementation of the LBM, which deals with geometries, by using curved boundaries-based methodologies, that is able to achieve a high performance on GPUs. Curved boundaries are taken into account via a non equilibrium extrapolation scheme developed in [13]. In [12], S. K. Layton et al. studied the solution of two-dimensional incompressible viscous flows with immersed boundaries using the *IB projection* method introduced in [14]. Their numerical framework is based on a Navier–Stokes solver and uses the `Cusp` library developed by Nvidia [15] for GPU acceleration. Our framework

* Corresponding author.
E-mail address: pedro.valero.lara@gmail.com (P. Valero-Lara).

uses a different Immersed Boundary formulation based on the one introduced by Uhlmann [16], which is able to deal with complex, moving or deformable boundaries [11,16–22]. Some previous performance results were presented in [23]. In this contribution we include a more elaborated discussion about performance results, and as a novelty, we explore alternative heterogeneous platforms based on the recently introduced Intel Xeon Phi device. Special emphasis is given to the implementation techniques adopted to mitigate the overhead of the immersed boundary correction and keep the solver highly efficient.

The remainder of this paper is organized as follows. In Section 2 we introduce the physical problem at hand and the general numerical framework that has been selected to cope with it. In Section 3 we review some optimizations of the baseline global LBM solver. After that, we describe the different optimizations and parallel strategies envisaged to achieve high-performance when introducing the IB correction. In Section 4 we focus on optimising this correction as it was a standalone kernel. Additional optimisations that take into account the interaction with the global LBM solver are studied afterwards in Section 5. Finally, we discuss the performance results of the proposed techniques in Section 6. We conclude in Section 7 with a summary of the main contributions of this work.

2. Numerical framework

As mentioned above, we have explored in this work a numerical framework based on the Lattice Boltzmann method coupled to the Immersed Boundary method. This combination is highly attractive when dealing with immersed bodies for two main reasons: (1) the shape of the boundary, tracked by a set of Lagrangian nodes is a sufficient information to impose the boundary values; and (2) the force of the fluid on the immersed boundary is readily available and thus easily incorporated in the set of equations that govern the dynamics of the immersed object. In addition, it is also particularly well suited for parallel architectures, as the time advancement is explicit and most computations are local [11]. In what follows, we briefly recall the basic formulation of LBM and then we describe the investigated LBM-IB framework.

2.1. The LBM method

Lattice Boltzmann has been extensively used in the past decades (see [24] for a complete overview) and today is widely accepted in both academia and industry as a powerful and efficient alternative to classical Navier Stokes solvers for simulating (time-dependent) incompressible flows [11].

LBM is based on an equation that governs the evolution of a discrete distribution function $f_i(\mathbf{x}, t)$ describing the probability of finding a particle at Lattice site $\mathbf{x}$ at time t with velocity $\mathbf{v} = \mathbf{e}_i$. In this work, we consider the *BGK* formulation [25] that relies upon an unique relaxation time τ toward the equilibrium distribution $f_i^{(eq)}$:

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) - f_i(\mathbf{x}, t) = -\frac{\Delta t}{\tau}\left(f(\mathbf{x}, t) - f_i^{(eq)}(\mathbf{x}, t)\right) + \Delta t F_i \tag{1}$$

The particles can move only along the links of a regular lattice defined by the discrete speeds ($e_0 = c(0, 0)$; $e_i = c(\pm 1, 0)$, $c(0, \pm 1)$, $i = 1, \ldots, 4$; $e_i = c(\pm 1, \pm 1)$, $c(\pm 1, \pm 1)$, $i = 5, \ldots, 8$ with $c = \Delta x/\Delta t$) so that the synchronous particle displacements $\Delta \mathbf{x}_i = \mathbf{e}_i \Delta t$ never take the fluid particles away from the Lattice. For the present study, the standard two-dimensional 9-speed Lattice *D2Q9* (Fig. 1) is used [26], but all the techniques that are presented in this work can be easily applied to three dimensional lattices.

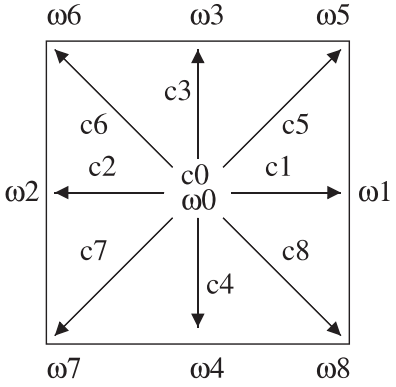


Fig. 1. Standard two-dimensional 9-speed lattice (*D2Q9*) used in our work.

The equilibrium function $f^{(eq)}(\mathbf{x}, t)$ can be obtained by Taylor series expansion of the Maxwell–Boltzmann equilibrium distribution [27]:

$$f_i^{(eq)} = \rho \omega_i \left[1 + \frac{\mathbf{e_i} \cdot \mathbf{u}}{c_s^2} + \frac{(\mathbf{e_i} \cdot \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}^2}{2c_s^2}\right] \tag{2}$$

In Eq. (2), c_s is the speed of sound ($c_s = 1/\sqrt{3}$), ρ is the macroscopic density, and the weight coefficients ω_i are $\omega_0 = 4/9$, $\omega_i = 1/9$, $i = 1, \ldots, 4$ and $\omega_5 = 1/36$, $i = 5, \ldots, 8$ according to the current normalization. The macroscopic velocity $\mathbf{u}$ in Eq. (2) must satisfy a Mach number requirement $|\mathbf{u}|/c_s \approx M \ll 1$. This stands as the equivalent of the CFL number for classical Navier Stokes solvers.

F_i in Eq. (1) represents the contribution of external volume forces at lattice level that in our case include the effect of the immersed boundary. Given any external volume force $\mathbf{f}_{ib}$, the contributions on the lattice are computed according to the formulation proposed in [13] as:

$$F_i = \left(1 - \frac{1}{2\tau}\right) \omega_i \left[\frac{\mathbf{e_i} - \mathbf{u}}{c_s^2} + \frac{\mathbf{e_i} \cdot \mathbf{u}}{c_s^4}\mathbf{e_i}\right] \cdot \mathbf{f_{ib}} \tag{3}$$

The multi-scale Chapman Enskog expansion of Eq. (1), neglecting terms of $O(\epsilon M^2)$ and using Eq. (3), returns the Navier–Stokes equations with body forces and the kinematic viscosity related to lattice scaling as $\nu = c_s^2(\tau - 1/2)\Delta t$.

Without the contribution of the external volume forces stemming from the immersed boundary treatment, Eq. (1) is advanced forward in time in two main stages, known as *collision* and *streaming*, as follows:

1. Calculation of the local macroscopic flow quantities ρ and $\mathbf{u}$ from the distribution functions $f_i(\mathbf{x}, t)$:

$$\rho = \sum f_i(\mathbf{x}, t) \text{ and } \rho \mathbf{u} = \sum \mathbf{e}_i f_i(\mathbf{x}, t)$$

2. Collision:

$$f_i^*(\mathbf{x}, t + \Delta t) = f_i(\mathbf{x}, t) - \frac{\Delta t}{\tau}\left(f(\mathbf{x}, t) - f_i^{(eq)}(\mathbf{x}, t)\right)$$

3. Streaming:

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i^*(\mathbf{x}, t + \Delta t)$$

A large number of researchers have investigated the performance aspects of this update process [8,11,28–32] over the past decade. In Section 3 we review these previous works and describe

the implementation techniques that we have opted to explore in our framework.

2.2. The LBM-IB framework

Next, we briefly introduce the Immersed Boundary method that we use both to enforce boundary values and to recover the fluid force exerted on immersed objects [16,22]. In the selected Immersed Boundary approach (as in several others), the fluid is discretized on a regular Cartesian lattice while the immersed objects are discretized and tracked in a Lagrangian fashion by a set of markers distributed along their boundaries. The general setup of the investigated Lattice Boltzmann-Immersed Boundary method can be recast in the following algorithmic sketch.

1. Given $f_i(\mathbf{x}, t)$ compute:

$$\rho = \sum f_i(\mathbf{x}, t) \text{ and } \rho\mathbf{u} = \sum \mathbf{e}_i f_i(\mathbf{x}, t) + \frac{\Delta t}{2}\mathbf{f}_{ib}$$

2. Collision stage:

$$\hat{f}_i^*(\mathbf{x}, t+\Delta t) = f_i(\mathbf{x}, t) - \frac{\Delta t}{\tau}\left(f(\mathbf{x}, t) - f_i^{(eq)}(\mathbf{x}, t)\right)$$

3. Streaming stage:

$$\hat{f}_i(\mathbf{x} + \mathbf{e}_i\Delta t, t+\Delta t) = \hat{f}_i^*(\mathbf{x}, t+\Delta t)$$

4. Compute:

$$\hat{\rho} = \sum \hat{f}_i(\mathbf{x} + \mathbf{e}_i\Delta t, t+\Delta t) \text{ and } \hat{\rho}\hat{\mathbf{u}} = \sum \mathbf{e}_i\hat{f}_i(\mathbf{x} + \mathbf{e}_i\Delta t, t+\Delta t)$$

5. Interpolate on Lagrangian markers (volume force):

$$\hat{\mathbf{U}}(\mathbf{X}_k, t+\Delta t) = \mathcal{I}(\hat{\mathbf{u}}) \text{ and } \mathbf{f}_{\mathbf{ib}}(\mathbf{x}, t)$$
$$= \frac{1}{\Delta t}\mathcal{S}\left(\mathbf{U}_d(\mathbf{X}_k, t+\Delta t) - \hat{\mathbf{U}}(\mathbf{X}_k, t+\Delta t)\right)$$

6. Repeat collision with body forces (see Eq. (3)) and Streaming:

$$f_i^*(\mathbf{x}, t+\Delta t) = f_i(\mathbf{x}, t) - \frac{\Delta t}{\tau}\left(f(\mathbf{x}, t) - f_i^{(eq)}(\mathbf{x}, t)\right)$$
$$+ \Delta t F_i \text{ and } f_i(\mathbf{x} + \mathbf{e}_i\Delta t, t+\Delta t) = f_i^*(\mathbf{x}, t+\Delta t)$$

As outlined above, the basic idea is to perform each time step twice. The first one, performed without body forces, predicts the velocity values at the immersed boundary markers and the force distribution that restores the desired velocity boundary values at their locations. The second one applies the regularized set of singular forces and repeats the procedure (using Eq. (3)) to determine the final values of the distribution function at the next time step. A key aspect of this algorithm is the way by which the interpolation $\mathcal{I}$ and the $\mathcal{S}$ operators (termed as spread from now on) are applied. Here, following [16,22], we perform both operations (interpolation and spread) through a convolution with a compact support mollifier meant to mimic the action of a Dirac's delta. Combining the two operators we can write in a compact form:

$$\mathbf{f}_{(ib)}(\mathbf{x}, t) = \frac{1}{\Delta t}\int_\Gamma \left(\mathbf{U_d}(\mathbf{s}, t+\Delta t) - \int_\Omega \hat{\mathbf{u}}(\mathbf{y})\tilde{\delta}(\mathbf{y}-\mathbf{s})\,d\mathbf{y}\right)\tilde{\delta}(\mathbf{x}-\mathbf{s})\,d\mathbf{s} \tag{4}$$

where $\tilde{\delta}$ is the mollifier, Γ is the immersed boundary, Ω is the computational domain, and $\mathbf{U_d}$ is the desired value on the boundary

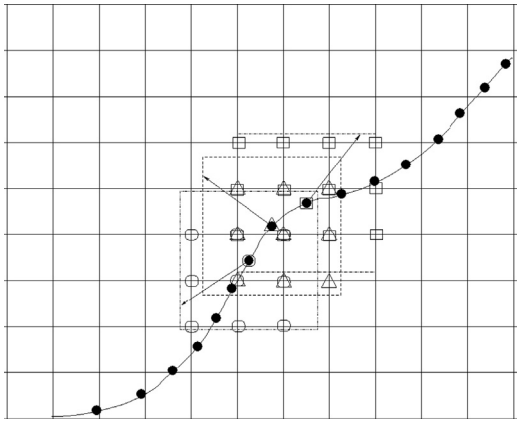


Fig. 2. An immersed curve discretized with *Lagrangian* points (•). Three consecutive points are considered with the respective supports.

at the next time step. The discrete equivalent of Eq. (4) is simply obtained by any standard composite quadrature rule applied on the union of the supports associated to each Lagrangian marker. As an example, the quadrature needed to obtain the force distribution on the lattice nodes is given by:

$$f_{ib}^l\left(x_i, y_j\right) = \sum_{n=1}^{N_e} F_{ib}^l(\mathbf{X_n})\,\tilde{\delta}\left(x_i - X_n, y_j - Y_n\right)\epsilon_n \tag{5}$$

where the superscript l refers to the lth component of the immersed boundary force, (x_i, y_j) are the lattice nodes (*Cartesian* points) falling within the union of all the supports, N_e is the number of Lagrangian markers and ϵ_n is a value to be determined to enforce consistency between interpolation and the convolution (Eq. (5)). More details about the method in general and the determination of the ϵ_n values in particular can be found in [22].

In what follows we will give more details on the construction of the support cages surrounding each Lagrangian marker since it plays a key role in the parallel implementation of the IB algorithm. Fig. 2 illustrates an example of the portion of the lattice units that falls within the union of all supports. As already mentioned, the embedded boundary curve is discretized into a number of markers $\mathbf{X}_I$, $I = 1, \ldots, N_e$. Around each marker $\mathbf{X}_I$ we define a rectangular cage Ω_I with the following properties: (i) it must contain at least three nodes of the underlying Eulerian lattice for each direction; (ii) the number of nodes of the lattice contained in the cage must be minimized. The modified kernel, obtained as a Cartesian product of the one dimensional function [33]

$$\tilde{\delta}(r) = \begin{cases} \frac{1}{6}\left(5 - 3|r| - \sqrt{-3(1-|r|)^2 + 1}\right) & 0.5 \le |r| \le 1.5 \\ \frac{1}{3}\left(1 + \sqrt{-3r^2 + 1}\right) & 0.5|r| \le 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

will be identically zero outside the square Ω_I. We take the edges of the square to measure slightly more than three lattice spacings Δ (i.e., the edge size is $3\Delta + \eta = 3 + \eta$ in the actual LBM normalization). With such choice, at least three nodes of the lattice in each direction fall within the cage. Moreover a value of $\eta \nmid 1$ ensures that the mollifier evaluated at all the nine (in two dimensions) lattice nodes takes on a non-zero value. The interpolation stage is performed locally on each nine points support: the values of velocity at the nodes within the support cage

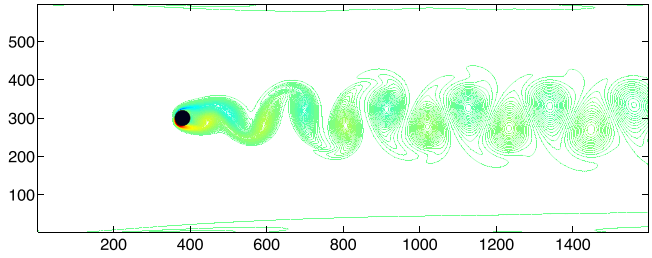


Fig. 3. Vorticity with Re = 100.

centered about each Lagrangian marker deliver approximate values (i.e., second order) of velocity at the marker location. The force spreading step requires information from all the markers, typically spaced $\Delta = 1$ apart along the immersed boundary. The collected values are then distributed on the union of the supports meaning that each support may receive information from supports centered about neighboring markers, as in Eq. (5). The outlined method has been validated for several test cases including moving rigid immersed objects and flexible ones [11].

2.3. Numerical validation of the method

Finally, we close this section presenting several test cases in order to validate the implementation of the code by comparing the numerical results obtained with other studies. One of the classical problems in CFD is the determination of the two-dimensional incompressible flow field around a circular cylinder, which is a fundamental problem in engineering applications. Several Reynolds numbers (20, 40 and 100) have been tested with the same configuration. The cylinder diameter D is equal to 40. The flow space is composed by a mesh equal to $40D\ (1600) \times 15D\ (600)$. The boundary conditions are set as: Inlet: $\mathbf{u} = U, \mathbf{v} = 0$, Outlet: $\frac{\partial \mathbf{u}}{\partial x} = \frac{\partial \mathbf{v}}{\partial x} = 0$, Upper and lower boundaries: $\frac{\partial \mathbf{u}}{\partial y} = 0, \mathbf{v} = 0$, Cylinder surface: $\mathbf{u} = 0, \mathbf{v} = 0$, Volume fraction: 0.5236%. When Reynolds number is 20 and 40, there is no vortex structure formed during the evolution. The flow field is laminar and steady. In contrast, for the Reynolds number of 100, the symmetrical rectangular zones disappear and an asymmetric pattern is formed. The vorticity is shed behind the circular cylinder, and vortex structures are formed downstream. This phenomenon is graphically illustrated in Fig. 3.

Two important dimensionless numbers are studied, the drag ($C_D = \frac{F_D}{0.5\rho U^2 D}$) and lift ($C_L = \frac{F_L}{0.5\rho U^2 D}$) coefficients. F_D corresponds to the resistance force of the cylinder to the fluid in the streamwise direction and F_L is the lifting force of the circular cylinder, ρ is the density of the fluid, and U is the velocity of inflow. In order to verify the numerical results, the coefficients were calculated and compared with the results of previous studies (Table 1). The drag coefficient for Reynolds number of 20 and 40 is equal to the results presented by Zhou et al. [9]. The drag coefficient obtained for Reynolds number of 100 is identical to the results obtained by Silva et al. [4], and the lift coefficient is close to the presented by Zhou et al. [9].

Table 1
Comparison between the numerical results yield by our method and previous studies.

Author	$Re = 20$	$Re = 40$	$Re = 100$	
	C_D	C_D	C_D	C_L
Calhoun [2]	2.19	1.62	1.33	0.298
Rusell [3]	2.22	1.63	1.34	–
Silva et al. [4]	2.04	1.54	1.39	–
Xu [1]	2.23	1.66	1.423	0.34
Zhou et al. [9]	2.3	1.7	1.428	0.315
This work	2.3	1.7	1.39	0.318

3. Implementation of the global LBM update

3.1. Parallelization strategies

Parallelism is abundant in the LBM update and can be exploited in different ways. On our GPU implementation, the lattice nodes are distributed across GPU cores using a fine-grained distribution. As shown in Fig. 4 (bottom), we used a 1D Grid of 1D CUDA Block. Each CUDA-thread performs the LBM update of a single lattice node. On multi-core processors, cache locality is a major performance issue and it is better to distribute the lattice nodes across cores using a 1D coarse-grained distribution (Fig. 4, top-left). The cache coherence protocol keeps the boundaries between subdomains updated. On the Intel Xeon Phi, we also distribute the lattice nodes across cores using a 1D coarse-grained distribution, but using a smaller block size (Fig. 4, top-right).

Another important issue is how to implement a single LBM update. Conceptually, a discrete time step consists of a local collision operation followed by a streaming operation that propagates the new information to the neighbour lattice nodes. However, most implementations do not apply those operations as separate steps. Instead, they usually fuse in a single loop nest (that iterates over the entire domain), the application of both operations to improve temporal locality [28,30].

This fused loop can be implemented in different ways. We have opted to use the pull scheme introduced in [28]. In this case, the body of the loop performs the streaming operation before collision, i.e. the distribution functions are gathered (pulled) from the neighbours before computing the collision. Algorithm 1 shows a sketch of our implementation.

Algorithm 1. LBM *pull*.

```
1:  Pull (f1 f2, ϖ, cx, cy)
2:  x, y
3:  x_stream, y_stream
4:  local_ux, local_uy, local_ρ
5:  local_f[9], f_eq
6:  for i = 1 → 9 do
7:      x_stream = x − c_x[i]
8:      y_stream = y − c_y[i]
9:      local_f[i] = f_1[x_stream][y_stream][i]
10: end for
11: for i = 1 → 9 do
12:     local_ρ += local_f[i]
13:     local_ux += c_x[i] × local_f[i]
14:     local_uy += c_y[i] × local_f[i]
15: end for
16: local_ux = local_ux / local_ρ
17: local_uy = local_uy / local_ρ
18: for i = 1 → 9 do
19:     f_eq = ω[i] · ρ · (1 + 3 · (c_x[i] · local_ux + c_y[i] · local_uy) +
        (c_x[i] · local_ux + c_y[i] · local_uy)^2 − 1.5 × ((local_ux)^2 + (local_uy)^2))
20:     f_2[x][y][i] = local_f[i] · (1 − 1/τ) + f_eq · 1/τ
21: end for
```

Other implementations [7,9,32,34] have used the traditional implementation sketched in Algorithm 2, which performs the collision operation before the streaming. It is known as the "push" scheme [28] since it loads the distribution function from the current lattice point and then it "pushes" (scatters) the updated values to its neighbours.

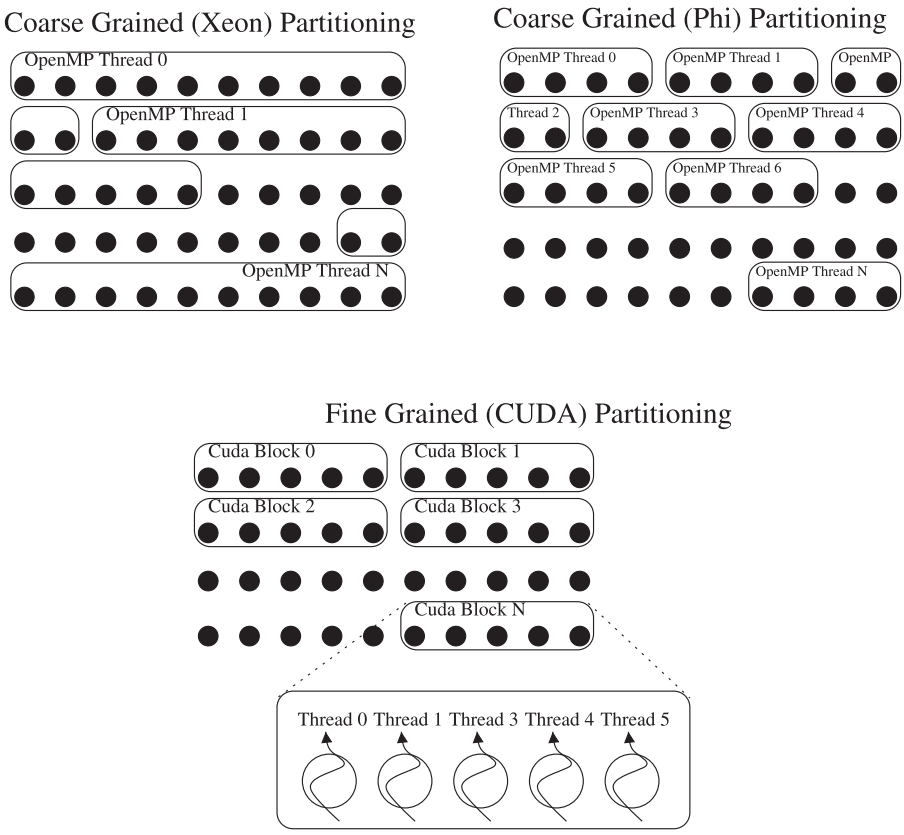


Fig. 4. Fine-grained and coarse-grained distributions of the lattice notes.

Algorithm 2. LBM *push*.

```
Collide Stream (u_x,u_y,ρ,f_1,f_2,ϖ,c_x,c_y)
x, y
x_stream, y_stream
f_eq
for i = 1 → 9 do
f_eq = ϖ[i] · ρ · (1 + 3 · (c_x[i] · u_x[x][y] + c_y[i] · u_y[x][y]) + (c_x[i] · u_x[x][y]
+ c_y[i] · u_y[x][y])^2 − 1.5 × ((u_x[x][y])^2 + (u_y[x][y])^2))
  f_1[x][y][i] = f_1[x][y][i] · (1 − 1/τ) + f_eq · 1/τ
  x_stream = x + c_x[i]
  y_stream = y + c_y[i]
  f_2[x_stream][y_stream][i] = f_1[x][y][i]
end for
Macroscopic (u_x,u_y,ρ,f_2,c_x,c_y)
x, y
local_ux,local_uy,local_ρ
for i = 1 → 9 do
  local_ρ += f_2[x][y][i]
  local_ux += c_x[i] × f_2[x][y][i]
  local_uy += c_y[i] × f_2[x][y][i]
end for
ρ[x][y] = local_ρ
u_x[x][y] = local_ux / local_ρ
u_y[x][y] = local_uy / local_ρ
```

3.2. Data layout and memory management

Since LBM is a memory-bound algorithm, another important optimization problem is to maximize data locality. Many groups have considered this issue and have introduced several data layouts and code transformations that are able to improve locality on different architectures [11,28–32,35,36]. Here we briefly describe the strategies used by our framework.

Different data structures have been proposed to store the discrete distribution functions f_i in memory:

- **AoS**. This data structure stores all the discrete distribution functions f_i of a given lattice point in adjacent memory positions (see Fig. 5(a)). This way, it optimizes locality when computing the collision operation [36]. However, it does not provide good performance on GPU architectures since it leads to poor bandwidth utilization [7,8,32].
- **SoA**. In this alternative data structure, the discrete distribution functions f_i for a particular velocity direction are stored sequentially in the same array (see Fig. 5(b)). Since each GPU thread handles the update of a single lattice node, consecutive GPU threads access adjacent memory locations with the SoA layout [7,8,32]. This way, they can be combined (coalesced) into a single memory transaction, which is not possible with the AoS counterpart.
- **SoAoS**. We have also explored a hybrid data structure, denoted as SoAoS in [36]. As SoA, it also allows coalesced memory transactions on GPUs. However, instead of storing the discrete distribution functions f_i for a particular velocity direction in a single sequential array, it distributes them across different blocks of a certain block size (see Fig. 5(c)). This size is a tunable parameter that trades off between spatial and temporal locality.

Apart from the data layout, the memory management of the different implementations may also differ in the number of lattices that are used internally. We have used a two-lattice implementation, which is denoted as the AB scheme in [29,32]. Essentially, AB holds the data of two successive time steps (A and B) and the simulation alternates between reading from A and writing to B, and vice versa. Other proposals, such as the AA data layout in [29,32], are able to use a single copy of the distributions arrays and reduce the memory footprint. Some previous works have shown that those single lattice schemes outperform the AB scheme on multi-core processors (AA achieved the best results in [31]). However, on GPUs

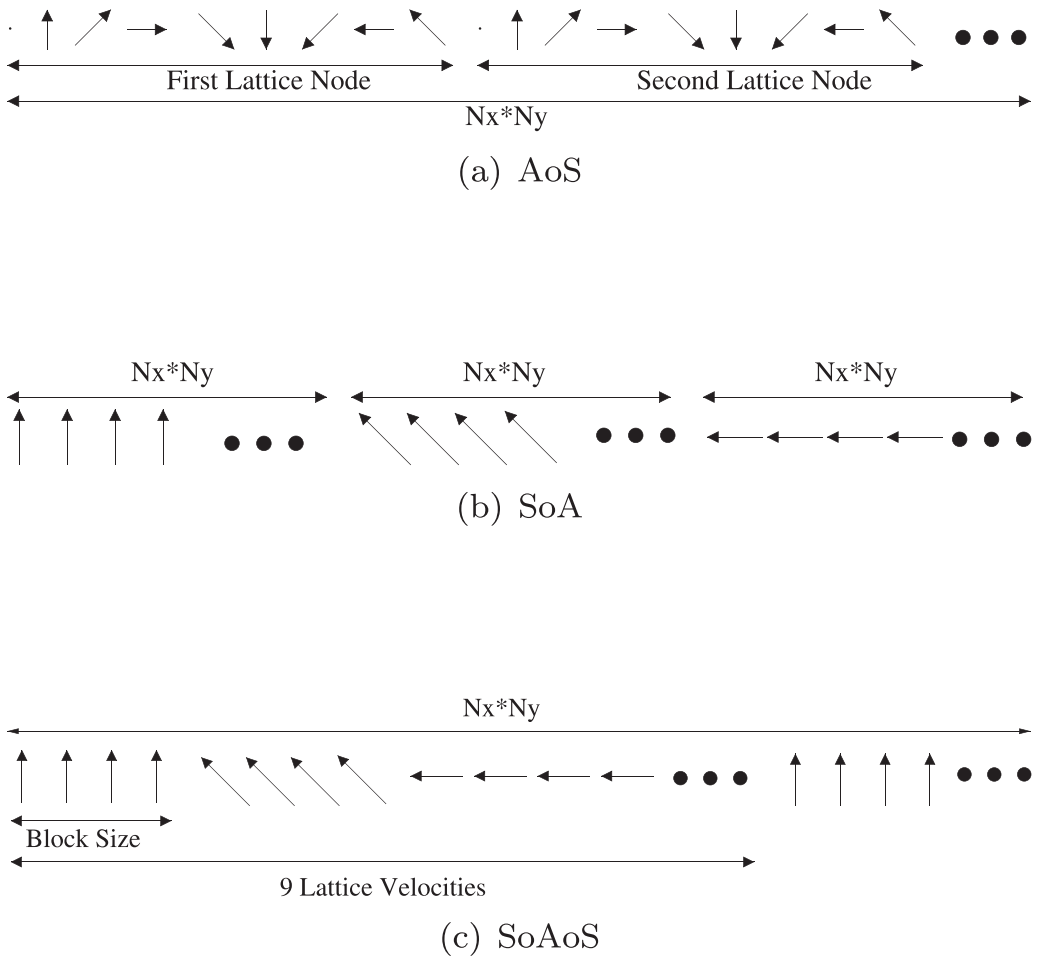


Fig. 5. Different data layouts to store the discrete distribution functions f_i in memory.

the performance benefits of these schemes are less clear. In fact, a recent work has shown that both schemes get similar performance on GPUs [32] or the AB scheme is typically a little faster on the latest GPUs. On the other hand, AB simplifies the integration with IB correction, which is the main focus of this research, and is therefore clearly preferred in our framework.

4. Implementation of the IB correction

In this section we discuss the different strategies that we have explored to optimize and parallelize the IB correction, i.e. the computations related with the *Lagrangian* markers distributed on the solid(s) surface(s).

4.1. Parallelization strategies

The parallelization of this correction step is only effective as long as there is sufficient work, which depends on the number of *Lagrangian* points in the solid surface and the overlapping among their supports.

On multi-core processors, we have been able to achieve satisfactory efficiencies (even for moderate number of points) using a coarse-grained distribution of (adjacent) *Lagrangian* points across all cores. This implementation is relatively straightforward using a few OpenMP pragmas that annotate the loops that iterate over the Lagrangian points.

On GPUs, it is better to exploit fine-grained parallelism. Our implementation consists of two main kernels denoted as IF (*Immersed Forces*) and BF (*Body Forces*) respectively. Both of them use the 1D distribution of threads across Lagrangian points shown in Fig. 6.

The IF kernel performs three major steps (see Algorithm 3). First, it assembles the velocity field on the supports. Then, it undertakes the interpolation at the *Lagrangian* markers and finally it determines the *Eulerian* volume force field on each node of the union of the supports. Note that forces are updated with *atomic* operations. Despite the overhead of such operations, they are necessary

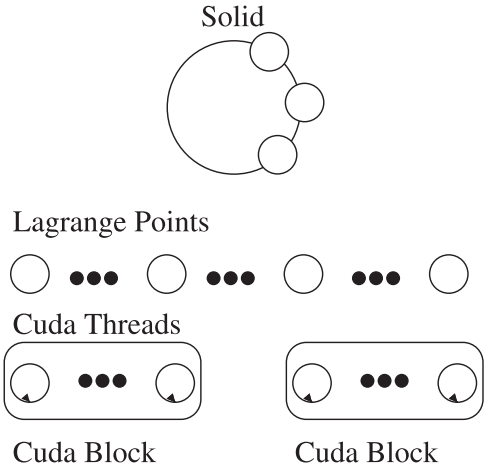


Fig. 6. 1D fine-grained distribution of *Lagrangian* points across CUDA threads.

to prevent race conditions since the supports of different *Lagrangian* points can share the same *Eulerian* points, as graphically shown in Fig. 2.

Once the IF kernel has been completed, the BF kernel (see Algorithm 4) computes the lattice forces and apply a local LBM update on the union of the supports, including now the contribution of the immersed boundary forces. Again, it avoids *race* conditions using *atomic* operations.

Algorithm 3. Pseudo-code of the IF kernel.

```
1:  IFC_kernel(solid s,U_x,U_y)
2:  vel_x, vel_y, force_x, force_y
3:  for i = 1 → numSupport do
4:    vel_x += interpol(U_x[s.Xsupp[i]], s)
5:    vel_y += interpol(U_y[s.Ysupp[i]], s)
6:  end for
7:  force_x = computeForce(vel_x, s)
8:  force_y = computeForce(vel_y, s)
9:  for i = 1 → numSupport do
10:   AddAtom(s.XForceSupp, spread(force_x, s))
11:   AddAtom(s.YForceSupp, spread(force_y, s))
12: end for
```

Algorithm 4. Pseudo-code of the BF kernel.

```
1:  BFC_kernel(solid s,f_x,f_y)
2:  F_body(Body Force),x, y,vel_x, vel_y
3:  for i = 1 → numSupport do
4:    x = s.Xsupport[i]
5:    y = s.Ysupport[i]
6:    vel_x = s.VelXsupport[i]
7:    vel_y = s.VelYsupport[i]
8:    for j = 1 →9 do
9:      F_body = (1 − 0.5 · 1/τ) · w[j] · (3 · ((c_x[j] − vel_x) · (f_x[x][y]) + (c_y[j] −
      vel_y) · f_y[x][y]))
10:     AddAtom(f^{t+1}[j][x][y], F_body)
11:   end for
12: end for
```

4.2. Data layout

Once again, another key aspect affecting performance in this correction step is the data structure (data layout) that is used to store the information about the *Lagrangian* points and their supports (coordinates, velocities and forces). Using the locality principle, it is natural to use an array of structures (AoS) to place all this information at a given *Lagrangian* point in nearby memory locations. This is the data structure that we have used on multi-core processors, since it optimizes cache performance. In contrast, on GPUs it is more natural to use a SoA data structure that distributes the information of all *Lagrangian* points in a set of one-dimensional arrays. With SoA, consecutive threads access to contiguous memory locations and memory accesses are combined (coalesced) in a single memory transaction. Fig. 7 illustrates the difference between those layouts in a simplified example with only two *Lagrangian* points.

5. Coupled LBM-IB on heterogeneous platforms

The complete LBM-IB framework can be fully implemented on many core processors combining the strategies described earlier for the global LBM updated and the IB correction. In this approach, the host processor stays idle most of the time and it is used exclusively for:

- **Pre-processing**. The host processor sets up the initial configuration and uploads those initial data to the main memory of the accelerator.
- **Monitoring**. The host processor is also in charge of a monitoring stage that downloads the information of each lattice node (i.e., velocity components and density) back to the host main memory when required.

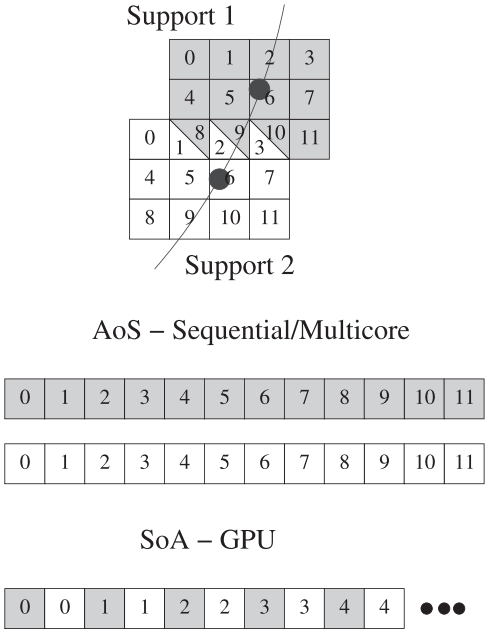


Fig. 7. Different data layouts used to store the information about the coordinates, velocities and forces of the *Lagrangian* points and their supports. On multi-core processors we use an AoS data structure (top) whereas on GPUs (bottom) we use a SoA data structure.

On GPUs, this approach consists of three main kernels, denoted in Fig. 8 as LBM, IF and BF respectively, which are launched consecutively for every time step. The first kernel implements the LBM update while the other two perform the IB correction. The overhead of the pre-processing stage performed on the multi-core processor has been experimentally shown to be negligible and the data transfer of the monitoring stage are mostly overlapped with the execution of the LBM kernel.

Although this approach achieves satisfactory results on GPUs, its speedups are substantially lower than those achieved by pure LBM solvers [8,32,29,11]. The obvious reason behind this behavior is the ratio between the characteristic volume fraction and the fluid field, which is typically very small. Therefore, the amount of data parallelism in the LBM kernel is substantially higher than in the other two kernels. In fact, for the target problems investigated, millions of threads compute the LBM kernel, while the IF and BF kernels only need thousands of them. But in addition, those kernels also require *atomic* functions due to the intrinsic characteristics of the IB method, and those operations usually degrade performance.

5.1. LBM-IB on hybrid multicore-GPU platforms

As an alternative to mitigate the overheads caused by the IB correction, we have explored hybrid implementations that take advantage of the host Xeon processor to hide them.

In Fig. 9 we show the hybrid implementation that we proposed on a previous paper [23] for heterogeneous multicore-GPU platforms. The LBM update is performed on the GPU as in the previous approach. However, IB and an additional local correction to LBM on the supports of the *Lagrangian* points are performed on the host processor. Both steps are coordinated using a pipeline. This way, we are able to overlap the prediction of the fluid field for the $t+1$ iteration with the correction of the IB method on the previous iteration t. This is possible at the expense of a local LBM computation of the"$t+1$" iteration on the multi-core and additional data transfers of the *supports* between the GPU and the host processor at each simulation step.

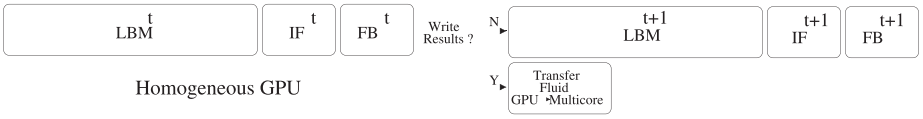


Fig. 8. Homogeneous GPU Implementation. Both the LBM update and the IB correction are performed on the GPU. The host processors stays idle most of the time.

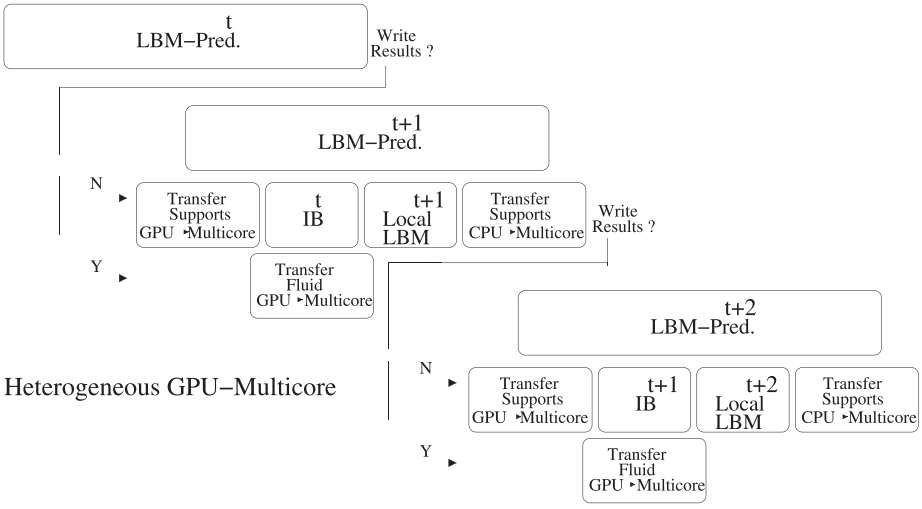


Fig. 9. Hybrid multicore-GPU implementation. The LBM update is performed on the GPU, whereas the IB correction and an additional step to update the supports of the *Lagrangian* points are performed on the multi-core processor.

5.2. LBM-IB on hybrid multicore-Xeon Phi platforms

The homogeneous Xeon Phi implementation suffers from the same performance problems that the homogeneous GPU counterpart. But again, multicore-Xeon Phi collaboration allows us to achieve higher performance. It is possible to use the same hybrid strategy introduced above for the multicore-GPU platform. However, we have opted to use a slightly different implementation that simplifies code development. Fig. 10 graphically illustrates the new partitioning. Essentially, it also consists on splitting the computational domain into two subdomains, so that one of them (*the solid subdomain*) fully includes the immersed solid. With such distribution, it is possible to perform the IB correction exclusively on the host Intel Xeon processor. However, in this case the host also performs the global LBM update on that subdomain, which simplifies the implementation.

As shown in Fig. 10, every time step, it is necessary to exchange the boundaries between both subdomains. To reduce the penalty of such data transfers, we update the boundaries between subdomains at the beginning of each time step. With this transformation, it is possible to exchange those boundaries with asynchronous operations that are overlapped with the update of the rest of the subdomain. In our target simulations, the *solid subdomain* is much smaller that the Xeon Phi counterpart. To improve performance, the size of both subdomains is adjusted to balance the loads between both processors.

6. Performance evaluation

6.1. Experimental setup

To critically evaluate the performance of the developed LBM-IB solvers, we have considered next a number of tests executed on two different heterogeneous platforms, whose main characteristics are summarized in Table 2.

On the GPU, the on-chip memory hierarchy has been configured as 16 KB shared memory and 48 KB L1, since our codes do not benefit from a higher amount of shared memory on the investigated tests.

Simulations have been performed using double precision, and we have used the conventional MFLUPS metric (millions of fluid lattice updates per second) to assess the performance.

6.2. Standalone Lattice-Boltzmann update

Before discussing the performance of our LBM-IB framework, it is important to determine the maximum performance that we can attain. We can estimate an upper limit omitting the IB correction.

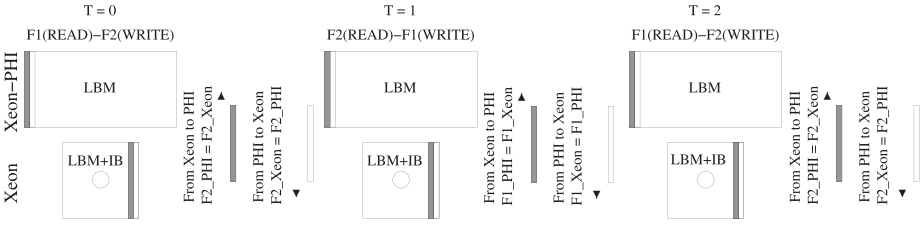


Fig. 10. Hybrid multicore-Phi implementation. The $L_x \times L_y$ lattice is split into two 2D sub-domains so that the IB correction is only needed on one of the domains. The multi-core processor updates the $L_xIB \times L_y$ subdomain, which fully includes the immersed solid (marked as a circle). The Xeon Phi updates the rest of the lattice nodes (the $L_xLBM \times L_y$ subdomain). The grey area highlights the ghost lattice nodes at the boundary between both sub-domains. In our target simulations, $L_xLBM \gg L_xIB$.

Table 2
Summary of the main features of the platforms used in our experimental evaluation.

Platform	Intel Xeon	NVIDIA GPU	Intel Xeon Phi
Model	E5520/E5-2670	Geforce GTX780 (Kepler)	5510P
Frequency	2.26/2.6 GHz	0.863	1.053 Ghz
Cores	8/16	2304	60
On-chip Mem.	L1 32 KB (per core)	SM 16/48 KB (per MP)	L1 32 KB (per core)
	L2 512 KB (unified)	L1 48/16 KB (per MP)	L2 256 KB (per core)
	L3 20 MB (unified)	L2 768 KB (unified)	L2 30 MB (coherent)
Memory	64/32GB DDR3	6GB GDDR5	8 GB GDDR5
Bandwidth	51.2 GB/s	288 GB/s	320 GB/s
Compiler	Intel Compiler 14.0.3	nvcc 6.5.12	Intel Compiler 14.0.3 [1]
Compiler Flags	-O3 -fomit-frame-pointer -fopenmp	-O3 -arch = sm_35	

[1] -O3 openmp -mcmodel=medium fno-alias -mP2OPT_hlo_use_const_pref_dist=64 -mP2OPT_hlo_use_const_second_pref_dist=32″

Fig. 11 shows the performance of this benchmark on a Kepler GPU. Recall from Section 3 that our implementation is based on the pull scheme and uses two lattice with the SoA data layout. As a reference we also show the performance of the *Sailfish* software package [32], which includes a LBM solver based on the *push* scheme, and the following estimation of the ideal MFLUPS [35]:

$$MFLUPS_{ideal} = \frac{B \times 10^9}{10^6 \times n \times 6 \times 8} \quad (7)$$

where $B \times 10^9$ is the memory bandwidth (GB/s), n depends on LBM model (DxQn), for our framework $n = 9$, D2Q9. The factor 6 is for the memory accesses, three read and write operations in the spreading step and three read and write operations in the collision step, and the factor 8 is for double precision (8 bytes).

Fig. 12(a) and (b) shows the performance on an Intel Xeon server. Although on multi-core processor it is natural to use the *AoS* data layout, *SoA* and *SoAoS* (with a block size of 32 elements) turn to be the most efficient data layouts. The main reason behind these unexpected results lies on the vector capabilities of modern processors. The compiler has been able to vectorize the main loops of the LBM update and both the AoS and SoAoS layouts allow a better exploitation of vectorization. Fig. 13 shows the scalability on this solver. The observed speedup factors over the sequential implementation almost peak with 16 threads, but the application scales relatively well since its performance is limited by the memory bandwidth.

Fig. 14(a) and (b) shows the performance on the Intel Xeon Phi. In this platform, *SoAoS* is able to outperform the other data layouts. The optimal SoAoS block size in this case is 128 elements, 4× the block size of the Intel Xeon, which coincides with the ratio between the vector widths of both architectures.

Fig. 15 analyses the impact of the thread-core affinity on performance using three different pre-defined strategies (see [37] for details on the rationale of each strategy). The *compact* affinity provides the best performance although the differences between them are not significant.

Finally, Fig. 16 shows the scalability on the Xeon Phi. For large problems, performance always improves as the number of threads increases. However, the gap with the ideal MFLUPS estimate is much larger in this case than in the other platforms. Given that the codes used for the Intel Xeon Phi and those used for the Intel

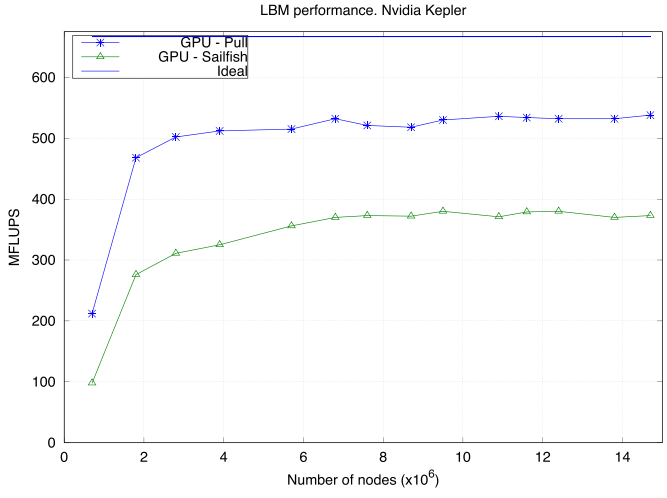
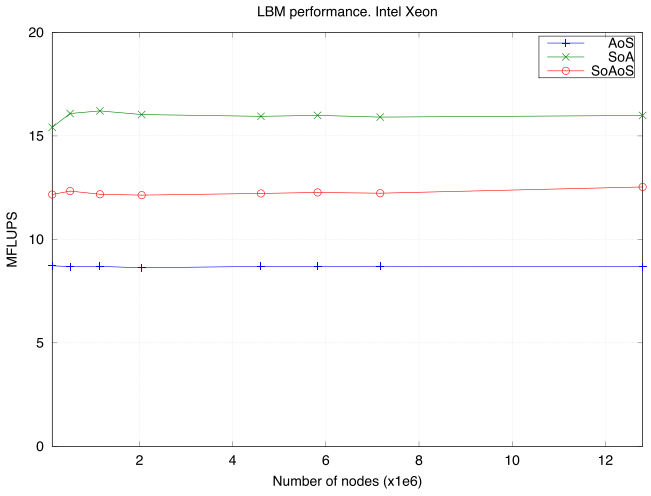
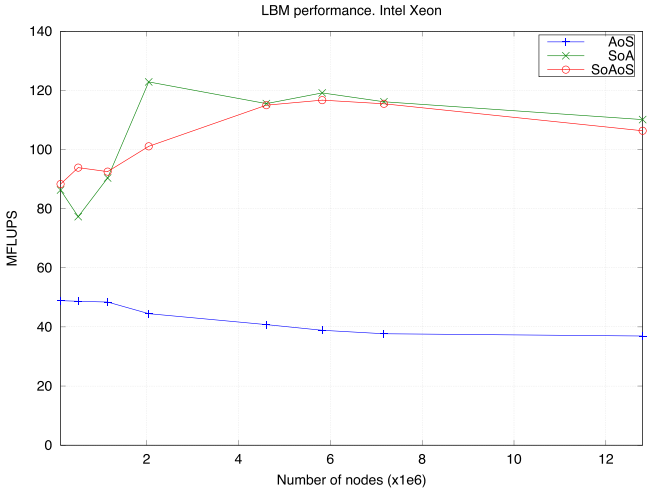


Fig. 11. Performance of the LBM update on the NVIDIA Kepler GPU.



(a) Sequential



(b) 32 threads

Fig. 12. Performance of the LBM update on the Intel Xeon multi-core processor.

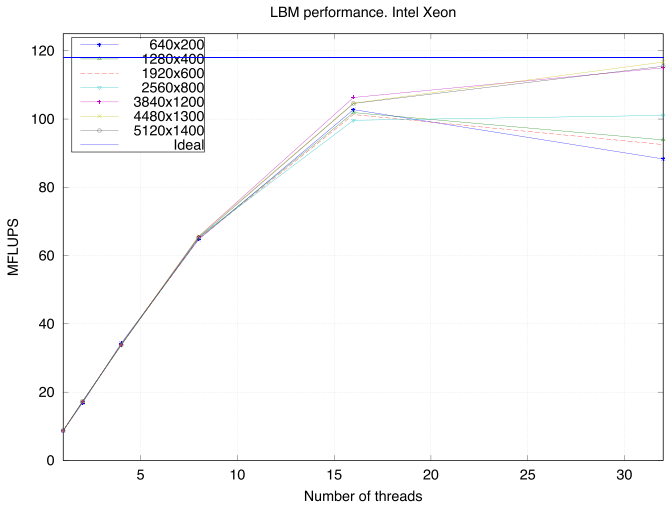
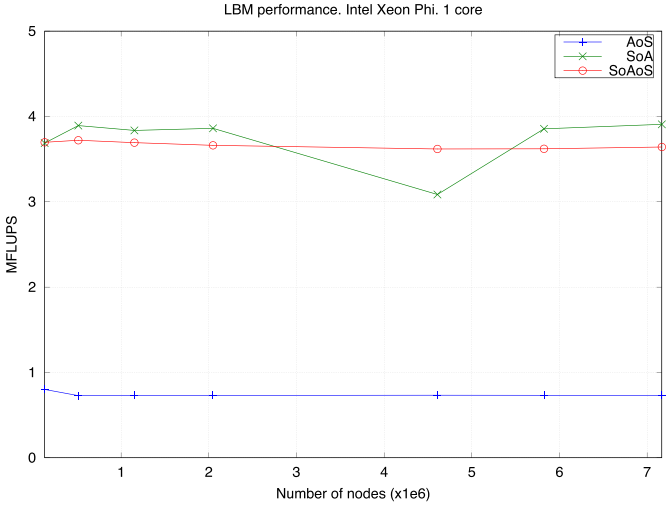
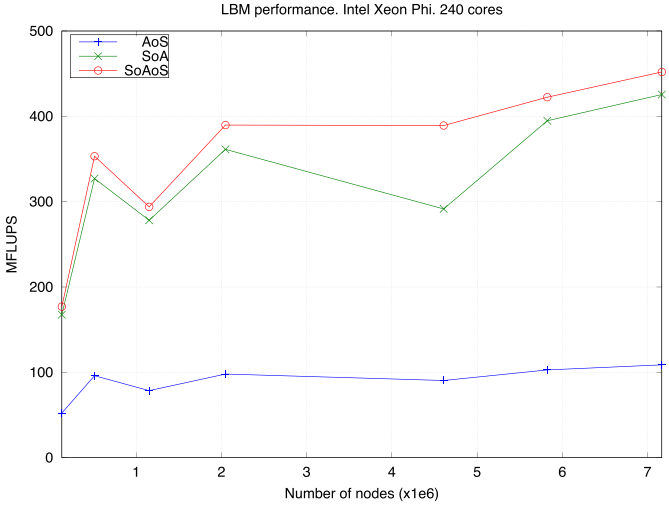


Fig. 13. Scalability of the LBM update on an Intel Xeon multi-core processor.



(a) Sequential



(b) 240 threads

Fig. 14. Performance of the LBM update on the Intel Xeon Phi.

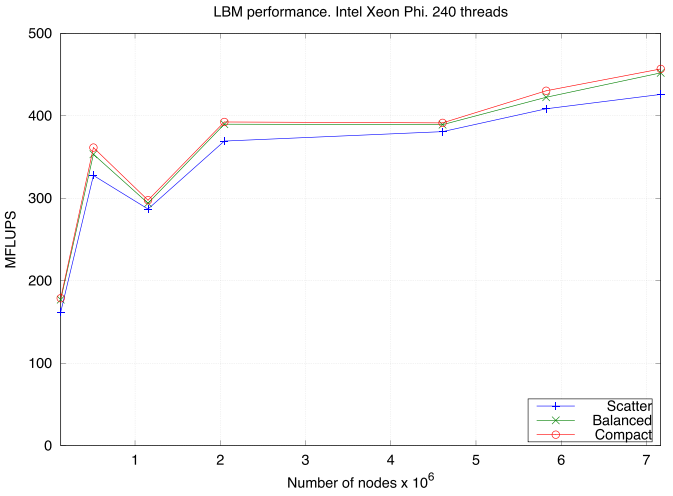


Fig. 15. Performance of the LBM update with different thread-core affinity strategies on the Intel Xeon Phi.

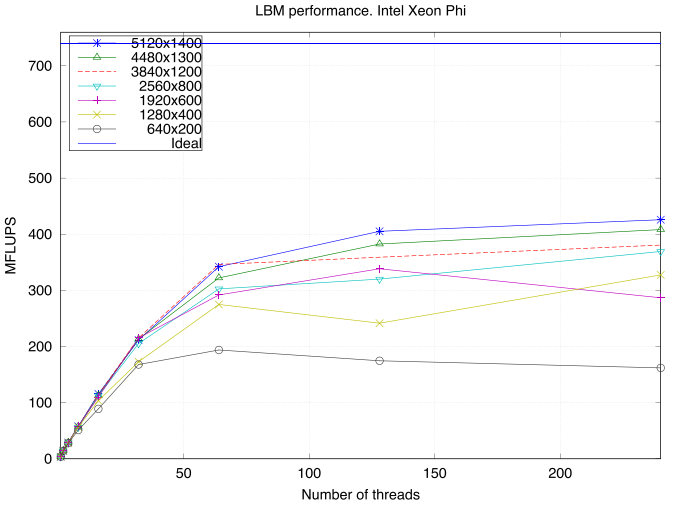


Fig. 16. Scalability of the LBM update on an Intel Xeon Phi.

Xeon are essentially the same (only optimal block sizes and minor optimization parameters change between both implementations) it is expected that either memory bandwidth is not limiting performance in this platform, or better performance values are expected for larger problem sizes.

Overall, the best performance is achieved on the GPU, which is able to outperform both the Intel Xeon and the Intel Xeon Phi. For the largest grid tested, the speed factor is 4.62× and 1.22×, respectively.

6.3. *Standalone IB correction*

Our second test uses a synthetic benchmark that focus exclusively on the IB correction, i.e., it omits the global LBM update and only applies the local IB correction. Fig. 17 shows the speedups of the parallel implementations over a sequential counterpart for increasing number of *Lagrangian* nodes. We are able to achieve substantial speedups, even for a moderate number of nodes. Notably, the best performance is achieved on the GPU, despite the overheads caused by the atomic updates needed on that implementation. From 2500 *Lagrangian* markers, the GPU outperforms the 8-core multicore counterpart with a 2.5× speedup.

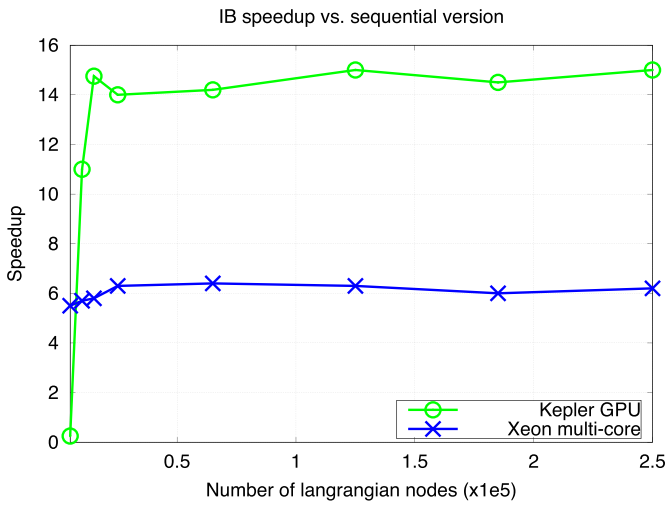


Fig. 17. Speedups of the IB method on multi-core and GPU for increasing number of Lagrangian nodes.

6.4. Coupled LBM-IB on heterogeneous platforms

6.4.1. Multicore-GPU

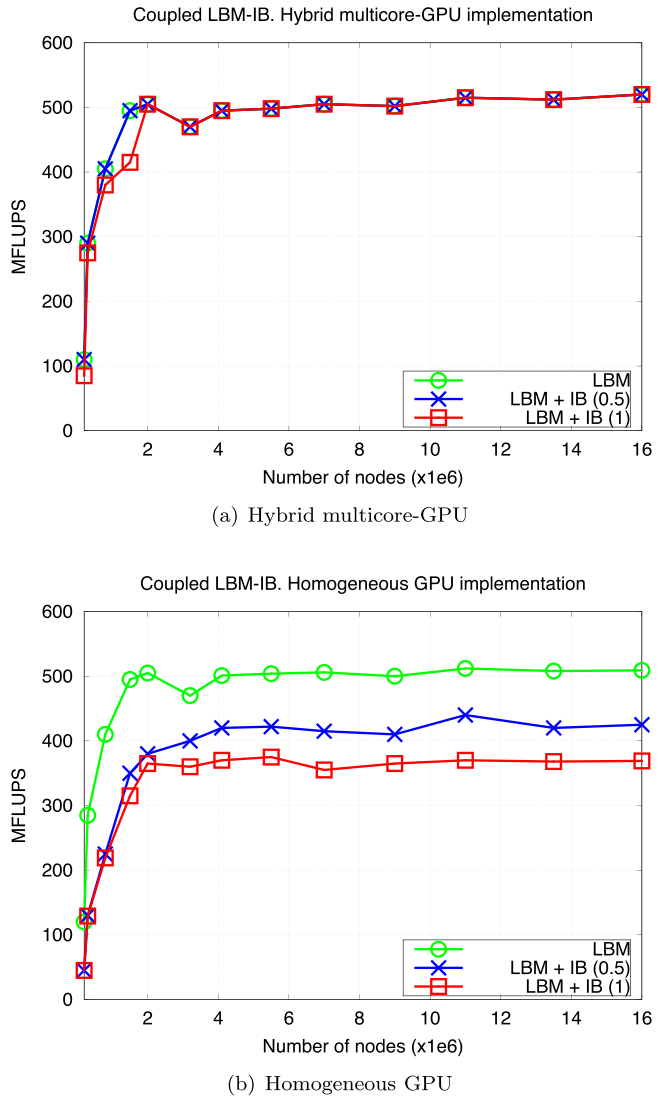
Fig. 18(a) shows the performance of the hybrid LBM-IB solver on a multicore-GPU heterogeneous platform for an increasing number of lattice nodes. As a reference, Fig. 18(b) shows the performance of the homogeneous GPU implementation. We have used the same physical setting studied in Section 2 and we have investigated two realistic scenarios with characteristic volume fractions of 0.5% and 1% respectively (i.e. the amount of embedded *Lagrangian* markers also grows with the number of lattice nodes).

The performance (MFLUPS) of the homogeneous implementation (Fig. 18(b)) drops substantially over the pure LBM implementation (Fig. 11). The slowdown is around 15% for a solid volume fraction of 0.5%, growing to 25% for the 1% case. In contrast, for these fractions the hybrid GPU-multicore version is able to hide the overheads of the IB method, having similar performance to the pure LBM implementation.

Fig. 19 shows a breakdown of the execution time for the simulation with a solid volume fraction of 1%. On the hybrid implementation, the cost of the IB correction is higher than in the homogeneous counterpart, reaching around 65% of the total execution time. This is expected since the in this case, the IB correction includes additional data transfers and local LBM updates. However, these costs are hidden with the global LBM update.

6.4.2. Multicore-Xeon Phi

Similarly, the performance of the proposed strategy over the multicore-Xeon Phi heterogeneous platform is analyzed in



(a) Hybrid multicore-GPU

(b) Homogeneous GPU

Fig. 18. Performance of the complete LBM-IB solver for an increasing number of lattice nodes.

Fig. 20(a). We have focused on the same numerical scenario described earlier with a solid volume fraction of 1%. As in the multicore-GPU platform, the observed performance in MFLUPS roughly matches the performance of the pure-LBM implementation on the same platform (Fig. 14(b)), with a peak performance of 450 MFLUPS for the largest problem size. $L_x IB$, which defines the size of the subdomain that is simulated on the multi-core

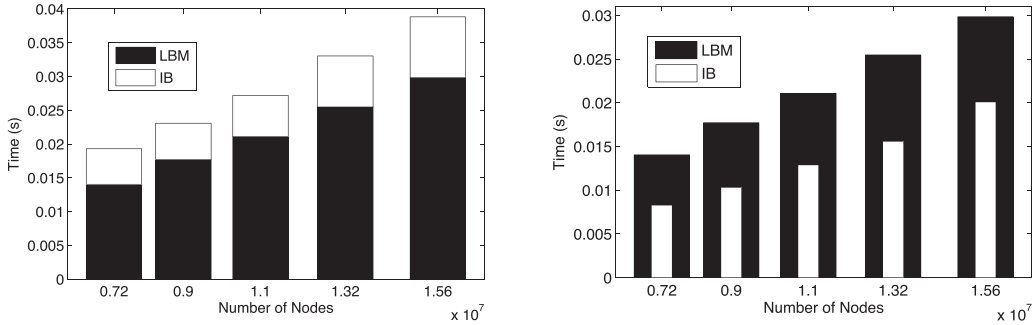
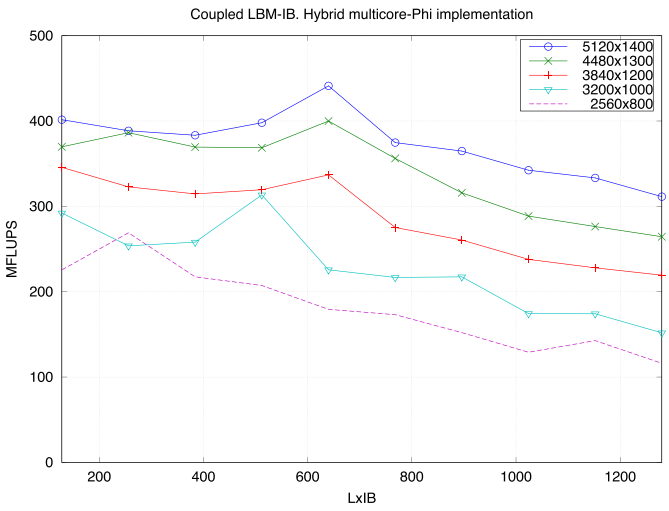
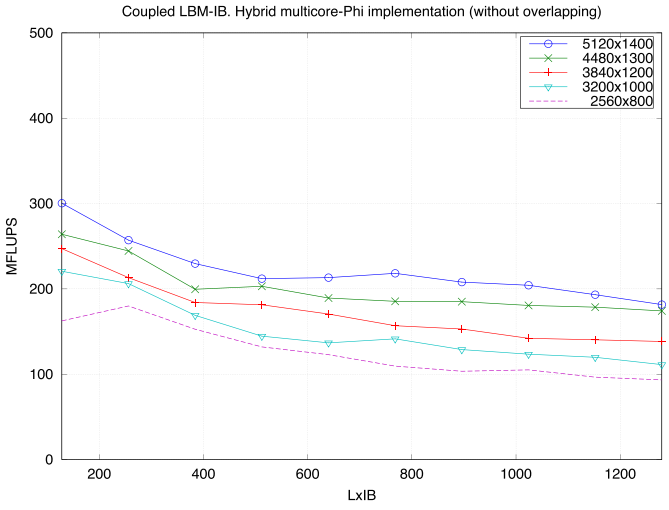


Fig. 19. Execution time breakdown for a solid volume fraction of 1% of the LBM and IB kernels on the homogeneous GPU implementation (left) and multicore-GPU heterogeneous (right) platforms.



(a) Hybrid multicore-Xeon Phi



(b) Hybrid multicore-Xeon Phi (without overlapping)

Fig. 20. Performance of the complete LBM-IB solver on the multicore-Phi platform as the size of subdomain that is simulated on the multi-core procesor increases.

processor, is a critical parameter in this implementation. As expected, there is an optimal value, which depends on the size of both the immersed solid and the grid, that balances the load in both processors. For example, the peak performance for the smallest grid tested ($L_x = 2560$, $L_y = 800$) is attained when the Xeon subdomain is roughly a 10% of the complete grid ($L_xIB = 250$), growing to 12% for the largest grid tested ($L_x = 5120$, $L_y = 1400$; ($L_xIB = 620$)).

Overall, as we have aimed, the overhead of the solid interaction is mainly hidden thanks to the effective cooperation with the multi-core processor. The penalty of the data transfers between both processors is negligible since boundary exchanges are conveniently orchestrated to allow their overlapping with useful computations.

Finally, as a reference, Fig. 20(b) shows the performance attained by an equivalent LBM-IB implementation, but without overlapping the execution on the Intel Xeon and the Intel Xeon Phi, i.e. the Xeon and the Phi subdomains are updated sequentially, one after the other. In this case, the peak performance drops to 300 MFLUPS, which highlights the benefit of overlapping the execution of both processors. Obviously, the lower L_xIB, the higher performance we achieve with this synthetic code since it increases the amount of work delivered to the Intel Xeon Phi.

7. Conclusions

In this paper, we have investigated the performance of a coupled Lattice-Boltzmann and Immersed Boundary method that simulates the contribution of solid behavior within an incompressible fluid. While LBM has been widely studied on heterogeneous platforms, the coupling of LBM with Immersed-Boundary methods has received less attention.

We have reviewed different strategies to enhance the performance on three state-of-the-art parallel architectures: an Intel Xeon server, a NVIDIA KeplerGPU and an Intel Xeon Phi accelerator. Our baseline LBM solver uses most of the state-of-the-art code transformations that have been described in previous work. Notably, performance results (MFLUPS) on the GPU and the multi-core processor are close to ideal MFLUPS estimations. On the Xeon Phi, the gap with these ideal estimations is higher, but it also achieves competitive performance. Overall, the best results are achieved on the GPU, which peaks at 550 MFLUPs, whereas the Intel Xeon Phi peaks at 450 MFLUPs.

Our main contribution is the design and analysis of a hybrid implementation that takes advantage of both the accelerator (GPU/Xeon Phi) and multi-core in a co-operative way to solve the coupled LBM-IB problem. For interesting physical scenarios with realistic solid volume fractions, these hybrid solvers are able to hide the overheads caused by the IB correction and match the performance (in terms of MFLUPS) of state-of-the-art pure LBM solvers. This has been possible thanks to the effective cooperation between the accelerator and the multi-core processor and the overlapping of data transfers between their memory spaces with useful computations.

Based on the presented LBM-IB framework, we envision two main research lines as future work. First, we will investigate an adaptation of the framework to problems which deal with deformable and moving bodies. In this type of scenarios, the work partitioning and assignment among computational resources will need to follow a dynamic approach instead of a static strategy as that implemented in this paper, to adapt the workload to the specific configuration at each timestep. Second, we plan to extend our work to 3D simulations; for these problems, memory consumption arises as one of the main problems, naturally driving to distributed-memory architectures, possibly equipped with hardware accelerators. In this case, we believe many of the techniques presented in this work will be of direct application at each node level; the scalability and parallelization strategies across nodes is still a topic under study.

Acknowledgments

This research is supported by the Spanish Government Research Contracts TIN2012-32180, Ingenio 2010 Consolider ESP00C-07-20811, MTM2013-40824 and SEV-2013-0323, by the EU-FET grant EUNISON 308874 and by the Basque government BERC 2014-2017 contract. We also thank the support of the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT) and NVIDIA GPU Research Center program for the provided resources.

References

[1] S. Xu, Z.J. Wang, An immersed interface method for simulating the interaction of a fluid with movies boundaries, J. Comput. Phys. 216 (2) (2006) 454–493.
[2] D. Calhoun, A cartesian grid method for solving the two-dimensional stream function-vorticity equations in irregular regions, J. Comput. Phys. 176 (2) (2002) 231–275, http://dx.doi.org/10.1006/jcph.2001.6970
[3] D. Russell, Z.J. Wang, A Cartesian grid method for modelling multiple moving objects in 2d incompressible viscous flows, J. Comput. Phys. 191 (2003) 177–205.

[4] A. Lima, E. Silva, A. Silveira-Neto, J.J.R. Damasceno, Numerical simulation of two-dimensional flows over a circular cylinder using the immersed boundary method, J. Comput. Phys. 189 (2) (2003) 351–370, http://dx.doi.org/10.1016/S0021-9991(03)00214-6

[5] P. Valero-Lara, A. Pinelli, J. Favier, M.P. Matias, Block tridiagonal solvers on heterogeneous architectures, in: Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ISPA'12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 609–616, http://dx.doi.org/10.1109/ISPA.2012.91

[6] P. Valero-Lara, A. Pinelli, M. Prieto-Matias, Fast finite difference Poisson solvers on heterogeneous architectures, Comput. Phys. Commun. 185 (4) (2014) 1265–1272, http://dx.doi.org/10.1016/j.cpc.2013.12.026

[7] M. Bernaschi, M. Fatica, S. Melchiona, S. Succi, E. Kaxiras, A flexible high-performance lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries, Concurr. Comput. Pract. Exp. 22 (2010) 1–14.

[8] P. Rinaldi, E. Dari, M. Vnere, A. Clausse, A lattice-boltzmann solver for 3d fluid simulation on {GPU}, Simul. Model. Pract. Theory 25 (0) (2012) 163–171, http://dx.doi.org/10.1016/j.simpat.2012.03.004

[9] H. Zhou, G. Mo, F. Wu, J. Zhao, M. Rui, K. Cen, {GPU} implementation of lattice Boltzmann method for flows with curved boundaries, Comput. Methods Appl. Mech. Eng. 225–228 (0) (2012) 65–73, http://dx.doi.org/10.1016/j.cma.2012.03.011

[11] C. Feichtinger, J. Habich, H. Kstler, U. Rude, T. Aoki, Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU-GPU clusters, Parallel Comput. 46 (0) (2015) 1–13, http://dx.doi.org/10.1016/j.parco.2014.12.003

[11] J. Favier, A. Revell, A. Pinelli, A lattice Boltzmann-immersed boundary method to simulate the fluid interaction with moving and slender flexible objects, J. Comput. Phys. 261 (0) (2014) 145–161, http://dx.doi.org/10.1016/j.jcp.2013.12.052

[12] A.K.S.K. Layton, L.A. Barbaa, cuIBM – a GPU-accelerated immersed boundary method, in: 23rd International Conference on Parallel Computational Fluid Dynamics (ParCFD), 2011.

[13] Z. Guo, C. Zheng, B. Shi, An extrapolation method for boundary conditions in lattice Boltzmann method, Phys. Fluids 14 (6) (2002) 2007–2010, http://dx.doi.org/10.1063/1.1471914

[14] K. Taira, T. Colonius, The immersed boundary method: a projection approach, J. Comput. Phys. 225 (2) (2007) 2118–2137.

[15] S. Dalton, N. Bell, L. Olson, M. Garland, Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012, available at: http://cusplibrary.github.io/

[16] M. Uhlmann, An immersed boundary method with direct forcing for the simulation of particulate flows, J. Comput. Phys. 209 (2) (2005) 448–476.

[17] C.S. Peskin, The immersed boundary method, Acta Numer. 11 (2002) 479–517.

[18] J. Wu, C. Aidun, Simulating 3d deformable particle suspensions using lattice Boltzmann method with discrete external boundary force, Int. J. Numer. Methods Fluids 62 (2010) 765–783.

[19] W.-X. Huang, S.J. Shin, H.J. Sung, Simulation of flexible filaments in a uniform flow by the immersed boundary method., J. Comput. Phys. 226 (2) (2007) 2206–2228.

[20] L. Zhu, C.S. Peskin, Interaction of two flapping filament in a flow soap film, Phys. Fluids 15 (2000) 1954–1960.

[21] L. Zhu, C.S. Peskin, Simulation of a flapping flexible filament in a flowing soap film by the immersed boundary method, Phys. Fluids 179 (2002) 452–468.

[22] U.P.A. Pinelli, I. Naqavi, J. Favier, Immersed-boundary methods for general finite-differences and finite-volume Navier–Stokes solvers, J. Comput. Phys. 229 (24) (2010) 9073–9091.

[23] P. Valero-Lara, A. Pinelli, M. Prieto-Matias, Accelerating solid–fluid interaction using lattice-Boltzmann and immersed boundary coupled simulations on heterogeneous platforms, Proc. Comput. Sci. 29 (0) (2014) 50–61, http://dx.doi.org/10.1016/j.procs.2014.05.005, 2014 International Conference on Computational Science.

[24] S. Succi, The Lattice Boltzmann Equation for Fluid Dynamics and Beyond (Numerical Mathematics and Scientific Computation), Numerical Mathematics and Scientific Computation, Oxford University Press, USA, 2001.

[25] E.G.P. Bhatnagar, M. Krook, A model for collision processes in gases. I: Small amplitude processes in charged and neutral one-component system, Phys. Rev. E 94 (1954) 511–525.

[26] C.K. Aidun, J.R. Clausen, Lattice-Boltzmann method for complex flows, Annu. Rev. Fluid Mech. 42 (1) (2010) 439–472, http://dx.doi.org/10.1146/annurev-fluid-121108-145519

[27] Y.H. Qian, D. D'Humires, P. Lallemand, Lattice BGK models for Navier–Stokes equation, Europhys. Lett. 17 (6) (1992) 479.

[28] G. Wellein, T. Zeiser, G. Hager, S. Donath, On the single processor performance of simple lattice boltzmann kernels, Comput. Fluids 35 (8–9) (2006) 910–919, http://dx.doi.org/10.1016/j.compfluid.2005.02.008, Proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science.

[29] P. Bailey, J. Myre, S. Walsh, D. Lilja, M. Saar, Accelerating lattice Boltzmann fluid flow simulations using graphics processors, in: International Conference on Parallel Processing, 2009. ICPP'09, 2009, pp. 550–557, http://dx.doi.org/10.1109/ICPP.2009.38

[30] J. Habich, C. Feichtinger, H. Kstler, G. Hager, G. Wellein, Performance engineering for the lattice Boltzmann method on GPGPUs: architectural requirements and performance results, Comput. Fluids 80 (0) (2013) 276–282, http://dx.doi.org/10.1016/j.compfluid.2012.02.013, Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[31] M. Wittmann, T. Zeiser, G. Hager, G. Wellein, Comparison of different propagation steps for lattice Boltzmann methods, Comput. Math. Appl. 65 (6) (2013) 924–935, http://dx.doi.org/10.1016/j.camwa.2012.05.002, Mesoscopic Methods in Engineering and Science.

[32] M. Januszewski, M. Kostur, Sailfish, A flexible multi-GPU implementation of the lattice Boltzmann method, Comput. Phys. Commun. 185 (9) (2014) 2350–2368, http://dx.doi.org/10.1016/j.cpc.2014.04.018

[33] C.S.P.A.M. Roma, M.J. Berger, An adaptive version of the immersed boundary method, J. Comput. Phys. 153 (1999) 509–534.

[34] M. Schnherr, K. Kucher, M. Geier, M. Stiebler, S. Freudiger, M. Krafczyk, Multi-thread implementations of the lattice Boltzmann method on non-uniform grids for CPUs and GPUs, Comput. Math. Appl. 61 (12) (2011) 3730–3743, http://dx.doi.org/10.1016/j.camwa.2011.04.012, Mesoscopic Methods for Engineering and Science – Proceedings of ICMMES-09 Mesoscopic Methods for Engineering and Science.

[35] A.G. Shet, S.H. Sorathiya, S. Krithivasan, A.M. Deshpande, B. Kaul, S.D. Sherlekar, S. Ansumali, Data structure and movement for lattice-based simulations, Phys. Rev. E 88 (2013) 013314, http://dx.doi.org/10.1103/PhysRevE.88.013314

[36] A.G. Shet, K. Siddharth, S.H. Sorathiya, A.M. Deshpande, S.D. Sherlekar, B. Kaul, S. Ansumali, On vectorization for lattice based simulations, Int. J. Mod. Phys. C 24 (2013) 40011, http://dx.doi.org/10.1142/S0129183113400111

[37] R.W. Green, OpenMP thread affinity control, 2013 https://software.intel.com/en-us/articles/openmp-thread-affinity-control